Rewriting Codes for Flash Memories

Eitan Yaakobi, Hessam Mahdavifar, Paul H. Siegel, Alexander Vardy, Jack K. Wolf,

arXiv:1210.7515v1 [cs.IT] 28 Oct 2012

***Abstract*—Flash memory is a non-volatile computer memory comprising blocks of cells, wherein each cell can take on q different values or *levels*. While increasing the cell level is easy, reducing the level of a cell can be accomplished only by erasing an entire block. Since block erasures are highly undesirable, coding schemes — known as *floating codes* (or *flash codes*) and *buffer codes* — have been designed in order to maximize the number of times that information stored in a flash memory can be written (and re-written) prior to incurring a block erasure.**

An $(n,k,t)_q$ flash code $\mathbb{C}$ is a coding scheme for storing k information bits in n cells in such a way that any sequence of up to t writes can be accommodated without a block erasure. The total number of available level transitions in n cells is $n(q-1)$, and the *write deficiency* of $\mathbb{C}$, defined as $\delta(\mathbb{C}) = n(q-1) - t$, is a measure of how close the code comes to perfectly utilizing all these transitions. In this paper, we show a construction of flash codes with write deficiency $O(qk\log k)$ if $q \geqslant \log_2 k$, and at most $O(k\log^2 k)$ otherwise.

An $(n,r,\ell,t)_q$ buffer code is a coding scheme for storing a buffer of r ℓ-ary symbols such that for any sequence of t symbols it is possible to successfully decode the last r symbols that were written. We improve upon a previous upper bound on the maximum number of writes t in the case where there is a single cell to store the buffer. Then, we show how to improve a construction by Jiang et al. that uses multiple cells, where $n \geqslant 2r$.

***Index Terms*—Coding theory, flash memories, flash codes, buffer codes.**

I. Introduction

Flash memories are, by far, the most important type of nonvolatile computer memory in use today. Flash devices are employed widely in mobile, embedded, and mass-storage applications, and the growth in this sector continues at a staggering pace.

A flash memory consists of an array of floating-gate ***cells***, organized into ***blocks*** (a typical block contains about 2^{20} cells). The level or "state" of a cell is a function of the amount of charge (electrons) trapped within it. In *multilevel flash cells*, voltage is quantized to q discrete threshold values; consequently the level of each cell can be modeled as an integer in the range $0,1,\ldots,q-1$. Nowadays, the parameter q itself can range from $q = 2$ (the conventional two-state case) up to $q = 16$ and it can reach even higher values [6]. The most conspicuous property of flash-storage technology is its inherent asymmetry between cell programming (charge placement) and cell erasing (charge removal). While adding charge to a single cell is a fast and simple operation, removing charge from a cell is very difficult. In fact, flash technology does not allow a single cell to be erased — rather, only entire blocks can be erased. Such ***block erasures*** are not only time-consuming, but also degrade the physical quality of the memory. For example, a typical block in a multilevel flash memory can tolerate only about 10^4 or even fewer erasures before it becomes unusable, and as such the lifetime and performance of the memory is highly correlated with the frequency of block erasure operations. Therefore, it is of importance to design coding schemes that maximize the number of times information stored in a flash memory can be written (and re-written) prior to incurring a block erasure.

Such coding schemes — known as ***floating codes*** (or ***flash codes***) and ***buffer codes*** — were recently introduced in [1], [8], [9]. Since then, several more papers on this subject have appeared in the literature [5], [10]–[12], [15], [19]. It should be pointed out that flash codes and buffer codes can be regarded as examples of memories with constrained source, which were described in [12]. Yet another example of such codes are the write-once memory (WOM) codes [2], [4], [17], that have been studied since the early 1980s. In fact, flash codes may be regarded as a generalization of WOM-codes. Slightly different and yet very related are the rank modulation codes [13], [14]. In rank modulation, the information is not stored according to the exact cell levels but rather by the cell permutation which is derived from the ordering of these levels.

An $(n,k,t)_q$ flash code $\mathbb{C}$ is a coding scheme for storing k information bits in n flash-memory cells, with q levels each, in such a way that any sequence of up to t writes can be accommodated without incurring a block erasure. In the literature on flash codes, a ***write*** is always a bit-write — that is, a change $0 \to 1$ or $1 \to 0$ in the value of one of the k information bits. Observe that in order to accommodate such a write, at least one of the n cells must transition from a lower level to a higher level (since a cell's level, determined by its charge, can only increase). On the other hand, the total number of available level transitions in n flash cells is $n(q-1)$. Thus, throughout this paper, we characterize the performance of a flash code $\mathbb{C}$ in terms of its ***write deficiency***, defined as $\delta(\mathbb{C}) = n(q-1) - t$. According to the foregoing discussion, $\delta(\mathbb{C})$ is a measure of how close $\mathbb{C}$ comes to perfectly utilizing all the available cell-level transitions: exactly one per write. The primary goal in designing flash codes can thus be expressed as *minimizing deficiency*.

What is the smallest possible write deficiency $\delta_q(n,k)$ for an $(n,k,t)_q$ flash code, and how does it behave asymptotically as the code parameters k and n get large? The best-known lower bound, due to Jiang, Bohossian, and Bruck [9], asserts that

$$\delta_q(n,k) \;\geqslant\; \frac{1}{2}(q-1)\min\{n,k-1\} \qquad (1)$$

How closely can this bound be approached by code construc-

E. Yaakobi, H. Mahdavifar, P.H. Siegel, A. Vardy, and J.K. Wolf are with the Department of Electrical and Computer Engineering, University of California at San Diego, La Jolla, CA 92093, U.S.A. (e-mail: {eyaakobi, hessam, psiegel, avardy, jwolf}@ucsd.edu).

tions? It appears that the answer to this question depends on the relationship between k and n. In this paper, we are concerned mainly with the case where both k and n are large, and n is much larger than k (in particular, $n \geqslant k^2$). In Section VI, we consider the case where k/n is a constant. At the other end of the spectrum, the case $k > n$ has been studied in [12].

The first construction of flash codes for large k was reported by Jiang and Bruck [10], [11]. In this construction, the k information bits are partitioned into $m_1 = k/k'$ subsets of k' bits each (with $k' \leqslant 6$) while the memory cells are subdivided into $m_2 \geqslant m_1$ groups of n' cells each. Additional memory cells (called ***index cells***) are set aside to indicate for each subset of k' bits which group of n' memory cells is used to store them. The deficiency of the resulting flash codes is $O(\sqrt{qn})$. Note that for $n \geqslant k$, the lower bound on write deficiency in (1) behaves as $\Omega(qk)$, and thus does not depend on n. Consequently, the gap between the Jiang-Bruck construction [11] and the lower bound could be arbitrarily large, especially when n is much larger than k.

In [19], a different construction of flash codes was proposed. These codes are based upon representing the n memory cells as a high-dimensional array, and achieve a write deficiency of $O(qk^2)$. Crucially, the deficiency of these codes does *not* depend on n. Nevertheless, there is still a significant gap between $O(qk^2)$ — which is the best currently known deficiency result — and the lower bound of $\Omega(qk)$.

In this paper, we present a new construction of flash codes which reduces the gap between the upper and lower bounds on write deficiency to a factor that is *logarithmic* in the number of information bits k. This result is arrived at in several stages. As a starting point, we use the "indexed" flash codes of Jiang and Bruck [11]. In Section IV, we develop new encoding and decoding procedures for such codes that eliminate the need for index cells in the Jiang-Bruck construction [11]. The write deficiency achieved thereby is $O(qk^2)$, which coincides with the main result of [19]. When the encoding procedure developed in Section IV reaches its limit, there are still potentially numerous unused cell-level transitions. In Section V, we show how to take advantage of these transitions in order to accommodate even more writes. To this end, we introduce a new indexing scheme, which is invoked only after the encoding method of Section IV reaches its limit. Thereupon, we extend this idea recursively, through $\lceil \log_2 k \rceil$ different indexing stages. This leads to a result, established in Theorem 4, stating that

$$\Omega(qk) \leqslant \delta_q(n,k) \leqslant O\big(\max\{q, \log_2 k\}\, k \log k\big) \qquad (2)$$

for all $n \geqslant k^2$, where the upper bound is achieved constructively by the flash codes described in Section V. In Section VI, we present and discuss constructions of flash codes for the case where the number of memory cells n is not significantly larger than the number of bits k.

The other type of codes we discuss in this paper are the buffer codes. An $(n, r, \ell, t)_q$ buffer code is a coding scheme for storing a buffer of r ℓ-ary symbols such that for any sequence of t symbol writes, it is possible to successfully decode the last r symbols that were written without a block erasure. Given a buffer of r ℓ-ary symbols that has to be stored in n q-ary cells, the goal is to maximize the number of writes t.

In Section VII, we formally define buffer codes. Then, we study two extreme cases where the number of cells is either one or very large. For the former case, Jiang et al. gave in [1], [10] a construction as well as an upper bound on the number of writes. Their construction works for $n = 1, \ell = 2$ and guarantees $t = \left\lfloor \frac{q}{2^{r-1}} \right\rfloor + r - 2$ writes. The upper bound stated in [1], [10] for $n = 1$ asserts that

$$t \leqslant \left\lfloor \frac{q-1}{\ell^r - 1} \right\rfloor \cdot r + \lfloor ((q-1) \bmod (\ell^r - 1) + 1) \rfloor .$$

We will show how to improve this bound such that for $q \geqslant \ell^r$,

$$t \leqslant \left\lfloor \frac{q - \ell^r}{\frac{1}{r} \sum_{d|r} \varphi(\frac{r}{d}) \ell^d} \right\rfloor + r,$$

where φ is Euler's φ function.

If the buffer is binary ($\ell = 2$) and the number of cells is significantly larger than the buffer size r, then a trivial upper bound on the number of writes t is $n(q-1)$. Jiang et al. showed in [1], [10] how to achieve $t = (q-1)(n-2r+1) + r - 1$ writes. Assume that $q = 2$, then the number of writes is $n - r$ and after the i-th write, the buffer is stored between cells $i+1$ and $i+r$. If $q > 2$, then the cell levels are used layer by layer, where first only levels zero and one are used, then one and two, and so on. In the transition from one layer to another, first the buffer is copied and stored in the new layer and then more writes are allowed. Thus, this construction allows $n - r$ writes on the first layer and $n - 2r + 1$ more writes in all other layers, so the total number of writes is $t = n - r + (q-2)(n-2r+1) = (q-1)(n-2r+1) + r - 1$. We will show how to improve this construction such that in every transition between layers, the buffer is stored cyclically in the cells and thus is not copied as before. This improves the number of writes to $(q-1)(n-r)$.

II. Preliminaries and Flash Codes Definition

Let us now give a precise definition of flash codes that were introduced in the previous section. We use $\{0,1\}^k$ to denote the set of binary vectors of length k, and refer to the elements of this set as ***information vectors***. The set of possible levels for each cell is denoted by $\mathcal{A}_q = \{0, 1, \ldots, q-1\}$ and thought of as a subset of the integers. The q^n vectors of length n over $\mathcal{A}_q$ are called ***cell-state vectors***. With this notation, any flash code $\mathbb{C}$ can be specified in terms of two functions: an encoding map $\mathcal{E}$ and a decoding map $\mathcal{D}$. The ***decoding map*** $\mathcal{D} : \mathcal{A}_q^n \to \{0,1\}^k$ indicates for each cell-state vector $\boldsymbol{x} \in \mathcal{A}_q^n$ the corresponding information vector. In turn, the ***encoding map*** $\mathcal{E} : \{0, 1, \ldots, k-1\} \times \mathcal{A}_q^n \to \mathcal{A}_q^n \cup \{\mathsf{E}\}$ assigns to every index i and cell-state vector $\boldsymbol{x} \in \mathcal{A}_q^n$, another cell-state vector $\boldsymbol{y} = \mathcal{E}(i, \boldsymbol{x})$ such that $y_j \geqslant x_j$ for all j and $\mathcal{D}(\boldsymbol{y})$ differs from $\mathcal{D}(\boldsymbol{x})$ *only* in the i-th position. If no such $\boldsymbol{y} \in \mathcal{A}_q^n$ exists, then $\mathcal{E}(i, \boldsymbol{x}) = \mathsf{E}$ indicating that block erasure is required. To bootstrap the encoding process, we assume that the initial state of the n memory cells is $(0, 0, \ldots, 0)$. Henceforth, iteratively applying the encoding map, we can determine how *any sequence* of transitions $0 \to 1$ or $1 \to 0$

in the k information bits maps into a sequence of cell-state vectors, eventually terminated by the block erasure. This leads to the following definition.

Definition. *An $(n,k)_q$ flash code $\mathbb{C}(\mathcal{D},\mathcal{E})$ **guarantees** t **writes** if for all sequences of up to t transitions $0\to 1$ or $1\to 0$ in the k information bits, the encoding map $\mathcal{E}$ does not produce the block erasure symbol* E*. If so, we say that $\mathbb{C}$ is an $(n,k,t)_q$ code, and define the **deficiency of** $\mathbb{C}$ as $\delta(\mathbb{C}) = n(q-1)-t$.*

In addition to this definition, we will also use the following terminology. Given a vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$ over $\mathcal{A}_q$, we define its ***weight*** as $\mathrm{wt}(\boldsymbol{x}) = x_1 + x_1 + \cdots + x_m$ (where the addition is over the integers), and its ***parity*** as $\mathrm{wt}(\boldsymbol{x}) \bmod 2$.

III. Two-Bit Flash Codes

In this section, we present a construction of flash codes that uses n q-ary cells to store $k = 2$ bits. In [9], a construction with these parameters was presented and was shown to be optimal. The construction we present in this section will be proved to be optimal as well and we believe that it is more intuitive.

In this construction, the leftmost and rightmost cells correspond to the first and second bit, respectively. When rewriting, assume the first bit changes its value, then the leftmost cell of level less than $q-1$ is increased by one level. Similarly, whenever the second bit changes its value, the rightmost cell of level less than $q-1$ is increased by one level. In general, the cell-state vector has the following form:

$$(q-1,\ldots,q-1,x_i,0,\ldots,0,x_j,q-1,\ldots,q-1),$$

where $0 < x_i, x_j \leqslant q-1$. This principle repeats itself until only one cell is left with level less than $q-1$. Then, this cell is used to store two bits according to its residue modulo 4. If this residue is $0,1,2,3$ then the value of the bits is $(v_1,v_2) = (0,0),(1,0),(0,1),(1,1)$, respectively. The construction is presented for odd values of q and we will discuss later how to modify it for even values as well. In the remainder of the paper, these maps are described algorithmically, using (C-like) pseudo-code notation.

Decoding map $\mathcal{D}_{2B}$: The input to this map is a cell-state vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$. The output is the corresponding two-bit information vector (v_1, v_2).

```
i1 = find_left_cell(y1, y2, ..., yn);
i2 = find_right_cell(y1, y2, ..., yn);
if(i2 == 0)   // all cells are full
{ v1 = q - 1(mod 2); v2 = ⌊((q - 1)(mod 4))/2⌋;}
if (i1 == i2) // there is only one non-full cell
{ v1 = y_{i1}(mod 2); v2 = ⌊(y_{i1}(mod 4))/2⌋;}
if (i1 != i2) // there are at least two non-full cells
{ v1 = y_{i1}(mod 2); v2 = y_{i2}(mod 2);}
```

Encoding map $\mathcal{E}_{2B}$: The input to this map is a cell-state vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$, and an index $j \in \{1,2\}$ of the bit that has changed. Its output is either a new cell-state vector $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)$ or the erasure symbol E.

```
(y1, y2, ..., yn) = (x1, x2, ..., xn);
i1 = find_left_cell(y1, y2, ..., yn);
i2 = find_right_cell(y1, y2, ..., yn);
if(i2 == 0) return E;
if (i1 == i2) // there is only one non-full cell
{ if(j == 2) a = 2;
  else a = j + 2·(y_{i1}(mod 2));
  if(y_{i1} + a > q - 1) return E;
  else { y_{i1} = y_{i1} + a; return;} }
y_{ij} = y_{ij} + 1;
if ((i2 - i1 == 1) ∧ (y_{ij} == q - 1))
{ v_{ij} = 0; v_{i3-j} = y_{i3-j}(mod 2);
  a = 2 · v2 + v1 - (y_{i3-j}(mod 4));
  if(a < 0)  y_{i3-j} = y_{i3-j} + 4 + x;
  else  y_{i3-j} = y_{i3-j} + a; }
```

The function **find_left_cell(**$y_1, y_2, \ldots, y_n$**)** finds the leftmost cell of level less than $q-1$ and if there is not such a cell then it returns $n+1$. Similarly, the function **find_right_cell(**$y_1, y_2, \ldots, y_n$**)** finds the rightmost cell of level less than $q-1$ and if there is not such a cell then it returns 0. The notation y_{i_j} stands for the variable y_{i_1} in case $j=1$, and y_{i_2} if $j=2$. The same rule applies to $y_{i_{3-j}}$. The symbol $\wedge$ stands for the logical operator "and". The next theorem proves the number of writes this construction guarantees.

Theorem 1. *If there are n q-level cells and q is odd, then the code $\mathbb{C}(\mathcal{D}_{2B},\mathcal{E}_{2B})$ guarantees at least $t = (n-1)(q-1) + \left\lfloor \frac{q-1}{2} \right\rfloor$ writes before erasing.*

Proof: As long as there is more than one cell of level less than $q-1$, the weight of the cell-state vector increases by one on each write. This may change only after at least $(n-1)(q-1)$ writes. Assume that there is only one cell of level less than $q-1$ after $s = (n-1)(q-1)+k$ writes, where $k \geqslant 0$, and call it the i-th cell. Starting this write, the different residues modulo 4 of the i-th cell correspond to the four possible two-bit information vector (v_1,v_2). Therefore, on the s-th write, we also need to increase the level of the i-th cell so it will correspond to the correct information vector on this write. For all succeeding writes, if the second bit changes then the i-th cell increases by two levels. If the first bit changes from 0 to 1 then the i-th cell increases by one level and otherwise by three levels. Therefore, if there are m more writes and $v_1 = 0$ then the i-th cell increases by at most $2m$ levels, and if there are m more writes and $v_1 = 1$ then the i-th cell increases by at most $2m+1$ levels.

Let us consider all possible values of k and the information vector (v_1,v_2) on the s-th write in order to calculate the number of guaranteed writes before erasing. Note that on the s-th write $(v_1+v_2) \equiv s \pmod 2$. Furthermore, since q is odd, the value of the bit that is written changes from one to zero because it reaches level $q-1$, and thus the other bit has value $k \pmod 2$.

1) Assume $k(\mathrm{mod}\ 4) = 0$, then $(v_1, v_2) = (0,0)$ and the level of the i-th cell does not increase on the s-th write. Since $v_1 = 0$, after m writes the cell increases by at most $2m$ levels. Hence, there are at least $\frac{q-1-k}{2}$ more writes and the total number of writes is at least
$$(n-1)(q-1)+k+\frac{q-1-k}{2} \geqslant (n-1)(q-1)+\frac{q-1}{2}.$$

2) Assume $k(\mathrm{mod}\ 4) = 1$, then $(v_1, v_2) = (1,0)$ or $(v_1, v_2) = (0,1)$. If $(v_1, v_2) = (1,0)$ then on the s-th write the i-th cell does not increase its level and after m writes its level increases by at most $2m+1$ levels. If $(v_1, v_2) = (0,1)$ then the i-th cell increases by one level and after m writes its level increases by at most $2m$ more levels. Hence, in both cases there are at least $\frac{q-2-k}{2}$ more writes. Together we get that the total number of writes is at least
$$(n-1)(q-1)+k+\frac{q-2-k}{2} \geqslant (n-1)(q-1)+\frac{q-1}{2}.$$

3) Assume $k(\mathrm{mod}\ 4) = 2$, then $(v_1, v_2) = (0,0)$ and the i-th cell increases by two levels on s-th write. Since $v_1 = 0$, after m more writes the cell increases by at most $2m$ levels and hence there are at least $\lfloor (q-1-(k+2))/2 \rfloor$ more writes, where $k \geqslant 2$. Therefore, the total number of write is at least
$$(n-1)(q-1)+k+\frac{q-3-k}{2} \geqslant (n-1)(q-1)+\frac{q-1}{2}.$$

4) Assume $k(\mathrm{mod}\ 4) = 3$, then $(v_1, v_2) = (1,0)$ or $(v_1, v_2) = (0,1)$. If $(v_1, v_2) = (1,0)$ then on the s-th write the i-th cell increases by two levels and after m more writes it increases by at most $2m+1$ levels. If $(v_1, v_2) = (0,1)$ then the i-th cell increases by three levels and after m more writes it increases by at most $2m$ more levels. Hence there are at least $\frac{q-4-k}{2}$ more writes, where $k \geqslant 3$. Thus, the total number of writes is at least
$$(n-1)(q-1)+k+\frac{q-4-k}{2} \geqslant (n-1)(q-1)+\frac{q-1}{2}.$$

In any case, the guaranteed number of writes is $(n-1)(q-1)+\left\lfloor \frac{q-1}{2} \right\rfloor$. ■

For even values of q, the construction is very similar. As long as there is more than one cell of level less $q-1$ we follow the same rules for the encoding. For the decoding, since $q-1$ is no longer even, the value of v_1 is the parity of the cells $1, \ldots, i_1$, where i_1 is the leftmost cell of value less $q-1$. The value of v_2 is the parity of the cells $i_2, i_2+1, \ldots, n$, where i_2 is the rightmost cell of value less $q-1$. If there is only one cell left, then it represents a value of two bits as before according to its residue modulo 4. If the the index of the last available cell is i then

$$v_1 = (i-1+y_i)(\mathrm{mod}2),$$
$$v_2 = ((n-i)+\lfloor (y_i(\mathrm{mod}4))/2 \rfloor)(\mathrm{mod}2).$$

Also, the last cell does not reach level $q-1$ so it is always possible to distinguish what the last cell is. We omit the tedious details as the proof is similar to the case where q is odd.

IV. Index-less Indexed Flash Codes

Our point of departure is the family of so-called *indexed flash codes*, due to Jiang and Bruck [11], that were briefly described in Section I. In this section, we eliminate the need for index cells — and, thus, the overhead associated with these cells — in the Jiang-Bruck construction [11]. This is achieved by "encoding" the indices into the order in which the cell levels are increased.

As in [11], we partition the n memory cells into m groups of n' cells each. However, while in [11] the value of n' is more or less arbitrary, in our construction $n' = k$. We henceforth refer to such groups of $n' = k$ cells as ***blocks*** (though they are not related to the *physical blocks* of floating-gate cells which comprise the flash memory). We will furthermore use, throughout this paper, the following terminology. We say that:

- a block is ***full*** if all its cells are at level $q-1$;
- a block is ***empty*** if all its cells are at level zero;
- a block is ***active*** if it is neither full nor empty;
- a block is ***live*** if it is not full (either active or empty).

In our construction, each block represents *exactly one bit*. This implies that the total number of blocks, given by $m = \lfloor n/k \rfloor$, must be at least k, which in turn implies $n \geqslant k^2$. If n is not divisible by k, the remaining cells are simply left unused. Finally, we also assume that either k is even or q is odd. If this is not the case, we can invoke the same construction with k replaced by $k+1$ (and the last bit permanently set to zero).

The key idea is that each block is used to encode not only the current value of the bit that it represents, but also *which* of the k bits it represents. The value of the bit is simply the parity of the block. The index of the bit is encoded in the *order* in which the levels of the k cells are increased. For example, if the block stores the i-th bit, first the level of the i-th cell in the block is increased from 0 to $q-1$ in response to the transitions $0 \to 1$ and $1 \to 0$ in the bit value. Then, the same procedure is applied to the $(i+1)$-st cell, the $(i+2)$-nd cell, and so on, with the indices $i+1, i+2, \ldots$ interpreted cyclically (modulo k). This process is illustrated in the following example.

Example 1. Suppose that $k = 4$ and $q = 3$. If a block represents the first bit, then its cell levels will transition from $(0,0,0,0)$ to $(2,2,2,2)$ in the following order:

$$(0000) \to (1000) \to (2000) \to (2100) \to (2200)$$
$$\to (2210) \to (2220) \to (2221) \to (2222)$$

On the other hand, for a block that represents the second bit, the corresponding cell-writing order is given by:

$$(0000) \to (0100) \to (0200) \to (0210) \to (0220)$$
$$\to (0221) \to (0222) \to (1222) \to (2222)$$

The cell-writing orders for blocks that represent the third and fourth bits are given, respectively, by

$$(0000) \to (0010) \to (0020) \to (0021) \to (0022)$$
$$\to (1022) \to (2022) \to (2122) \to (2222)$$

and

$$(0000) \to (0001) \to (0002) \to (1002) \to (2002)$$

$$\to (2102) \to (2202) \to (2212) \to (2222)$$

Note that, unless a block is full, it is always possible to determine which cell was written first and, consequently, which of the $k = 4$ bits this block represents.

We now provide a precise specification of an $(n,k)_q$ flash code $\mathbb{C}$ based upon this idea, in terms of a decoding map $\mathcal{D}_0$ and an encoding map $\mathcal{E}_0$.

Decoding map $\mathcal{D}_0$: The input to this map is a cell-state vector $\boldsymbol{x} = (\boldsymbol{x}_1|\boldsymbol{x}_2|\cdots|\boldsymbol{x}_m)$, partitioned into m blocks. The output is the corresponding information vector $(v_0, v_1, \ldots, v_{k-1})$.

```
(v_0, v_1, ..., v_{k-1}) = (0, 0, ..., 0);
for (j = 1; j ⩽ m; j = j + 1)
if (active(x_j))
{ i = read_index(x_j); v_i = parity(x_j); }
```

Encoding map $\mathcal{E}_0$: The input to this map is a cell-state vector $\boldsymbol{x} = (\boldsymbol{x}_1|\boldsymbol{x}_2|\cdots|\boldsymbol{x}_m)$, partitioned into m blocks of k cells, and an index i of the bit that has changed. Its output is either a cell-state vector $\boldsymbol{y} = (\boldsymbol{y}_1|\boldsymbol{y}_2|\cdots|\boldsymbol{y}_m)$ or the erasure symbol E.

```
(y_1|y_2|···|y_m) = (x_1|x_2|···|x_m);
for (j = 1; j ⩽ m; j = j + 1)
if (active(x_j) ∧ (read_index(x_j) == i))
{ write(y_j); break; }
if (j == m + 1) // active block not found
for (j = 1; j ⩽ m; j = j + 1)
if (empty(x_j)) { write_new(i, y_j); break; }
if (j == m + 1) // no empty blocks remain
return E;
```

To complete the specification of the flash code $\mathbb{C}(\mathcal{D}_0, \mathcal{E}_0)$, let us elaborate upon all the functions used in the pseudo-code above. The function **`active(`$\boldsymbol{x}$`)`**, respectively **`empty(`$\boldsymbol{x}$`)`**, simply determines whether the given block is active, respectively empty. The function **`parity(`$\boldsymbol{x}$`)`** computes the parity of $\boldsymbol{x}$, defined in Section II. Note that the parity of a full block is always zero (since $k(q-1)$ is even, by assumption). The function **`read_index(`$\boldsymbol{x}$`)`** computes the bit-index encoded in an active block $\boldsymbol{x} = (x_0, x_1, \ldots, x_{k-1})$. This can be done as follows. Find all the zero cells in $\boldsymbol{x}$. Note that these cells always form one cyclically contiguous run, say $x_j, x_{j+1}, \ldots, x_{j+r}$ (where the indices are modulo k). Then the index of the corresponding bit is $i = j + r + 1 \pmod{k}$. If there are no zeros in $\boldsymbol{x}$, there must be exactly one cell, say x_j, whose level is strictly less than $q-1$. In this case, the bit-index is $i = j + 1 \pmod{k}$. The function **`write(`$\boldsymbol{y}$`)`** proceeds along similar lines. Find the single cyclically contiguous run of zeros in $(y_0, y_1, \ldots, y_{k-1})$, say $y_j, y_{j+1}, \ldots, y_{j+r}$. If $y_{j-1} < q-1$, increase y_{j-1} by one; otherwise set $y_j = 1$. If there are no zeros in $\boldsymbol{y}$, find the unique cell y_j such that $y_j < q-1$ and increase its level by one. Finally, the function **`write_new(`i`,`$\boldsymbol{y}$`)`** simply sets $y_i = 1$.

Theorem 2. *The write deficiency of the flash code $\mathbb{C}(\mathcal{D}_0, \mathcal{E}_0)$ described above is at most*

$$(k-1)\Big((k+1)(q-1) - 1\Big) = O(qk^2) \tag{3}$$

Proof. Note that at each instance, at most k of the m blocks are active. The encoding map $\mathcal{E}_0(i, \boldsymbol{x})$ produces the symbol E when there are no more empty blocks, and none of the active blocks represents the i-th bit. In the worst case, this may occur when there are $k-1$ active blocks, each using just one cell level. This contributes $(k-1)\big(k(q-1)-1\big)$ unused cell levels. In addition, there are at most $k-1$ cells that are unused due to the partition into $m = \lfloor n/k \rfloor$ blocks of exactly k cells. These contribute at most $(k-1)(q-1)$ unused cell levels. ∎

V. Nearly Optimal Construction

It is apparent from the proof of Theorem 2 that the deficiency of the flash code $\mathbb{C}(\mathcal{D}_0, \mathcal{E}_0)$, constructed in Section IV, is due primarily to the following: when writing stops, there may remain potentially large amount of unused cell levels. The key idea developed in this section is to *continue writing* after the encoding map $\mathcal{E}_0$ produces the erasure symbol E, utilizing those cell levels that are left unused by $\mathcal{E}_0$. Obviously, it is *not* possible to continue writing using the same encoding and decoding maps. However, it may be possible to do so if, at the point when $\mathcal{E}_0$ produces the erasure symbol E, we switch to a *different encoding procedure*, say $\mathcal{E}_1$. In fact, this idea can be applied iteratively: once $\mathcal{E}_1$ reaches its limit, we will transition to another encoding map $\mathcal{E}_2$, then yet another map $\mathcal{E}_3$, and so on.

Assuming that $k \equiv 0 \pmod 4$, here is one way to continue writing after the encoding map $\mathcal{E}_0$ has been exhausted. When $\mathcal{E}_0$ produces the erasure symbol E, we say that the *first stage* of encoding is over and transition to the *second stage*, as follows. First, we re-examine the cell-state vector $\boldsymbol{x} = (\boldsymbol{x}_1|\boldsymbol{x}_2|\cdots|\boldsymbol{x}_m)$ and re-partition it into $2m = 2\lfloor n/k \rfloor$ blocks of $k/2$ cells each. Most of these smaller blocks will already be full, but we may find some m_1 of them that are either empty or active (live). Observe that $m_1 \leqslant 2(k-1)$ since at the end of the first stage, there are at most $k-1$ active blocks of k cells, and each of them produces at most two live (non-full) blocks of $k/2$ cells.

If $m_1 \geqslant k$, we can continue writing as follows. Once again, each of the m_1 blocks will represent exactly one bit; as before, the value of this bit is determined by the parity of the block. As part of the transition from the first stage to the second stage, we record the current information vector $(v_0, v_1, \ldots, v_{k-1})$ in the first k of the m_1 live blocks, say $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k$. To this end, whenever **`parity(`$\boldsymbol{x}_i$`)`** $\neq v_{i-1}$, we increase the level of one of the cells in $\boldsymbol{x}_i$ by one; otherwise, we leave $\boldsymbol{x}_i$ as is.

Since the blocks now have $k/2$ cells rather than k cells, it is no longer possible to encode in each block *which* of the k information bits it represents. Therefore, we set aside for this purpose $2(k-1)\lceil \log_q(k+2) \rceil$ *index cells* (that are not used during the first stage). These cells are partitioned into $2(k-1)$ blocks of $\mu = \lceil \log_q(k+2) \rceil$ cells each, which we call ***index blocks***. Henceforth, it will be convenient to refer to the blocks of $k/2$ cells as ***parity blocks***, in order to distinguish them from the index blocks. Initially, the first k

index blocks $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_k$ are set so that $\boldsymbol{u}_i = i$ (in the base-q number system), which reflects the fact that the information bits $v_0, v_1, \ldots, v_{k-1}$ are stored (in that order) in the first k live parity blocks. The next $m_1 - k$ index blocks are set to $(0, 0, \ldots, 0)$, thereby indicating that the corresponding (live) parity blocks are available to store information bits. The last $2(k-1) - m_1$ index blocks are set to $(q-1, q-1, \ldots, q-1)$ to indicate that the corresponding parity blocks are full (in fact, nonexistent). Finally, it is possible that in the process of enforcing $\texttt{parity}(\boldsymbol{x}_i) = v_{i-1}$ for the first k live parity blocks, some of these blocks become full (this happens iff $\mathrm{wt}(\boldsymbol{x}_i) = (k/2)(q-1) - 1$ and $v_i = 0$ at the end of the first stage, since $k/2$ is even by assumption). To account for this fact, we set the corresponding index blocks to $(q-1, q-1, \ldots, q-1)$. This completes the transition from the first stage to the second stage, which is invoked when the encoding map $\mathcal{E}_0$ produces the erasure symbol E.

Let us now summarize the foregoing discussion by giving a concise algorithmic description of the transition procedure.

Transition procedure $\mathcal{T}_1$: Partition the memory into $2\lfloor n/k \rfloor$ parity blocks of $k/2$ cells, and identify the $m_1 \leqslant 2(k-1)$ parity blocks $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_{m_1}$ that are not full. If $m_1 < k$, output the erasure symbol E and terminate. Otherwise, set the $2(k-1)$ index blocks $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_{2k-2}$ as follows:

$$\boldsymbol{u}_i = \begin{cases} i & \text{for } i = 1, 2, \ldots, k \\ 0 & \text{for } i = k+1, k+2, \ldots, m_1 \\ q^\mu - 1 & \text{for } i = m_1{+}1, m_1{+}2, \ldots, 2k-2 \end{cases} \tag{4}$$

where $\mu = \lceil \log_q(k{+}2) \rceil$ is the number of cells in each index block, then record the information vector $(v_0, v_1, \ldots, v_{k-1})$ in the first k live parity blocks $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k$, as follows:

```
for (i = 1; i ⩽ k; i = i + 1)
if (parity(x_i) ≠ v_{i-1})
{ increment(x_i); if (full(x_i)) u_i = q^μ − 1;}
```

The function $\texttt{full}(\boldsymbol{x})$ determines whether the given block $\boldsymbol{x}$ (which could be a parity block or an index block) is full. The function $\texttt{increment}(\boldsymbol{x})$ increases by one the level of a cell (does not matter which) in the given live block.

During second-stage encoding and decoding, we will need to figure out for each active parity block $\boldsymbol{x}$ which of the k information bits it represents. To this end, we will have to find and read the index block $\boldsymbol{u}$ that *corresponds* to $\boldsymbol{x}$. How exactly is the correspondence between parity blocks and index blocks established? Note that, upon the completion of the transition procedure $\mathcal{T}_1$, there is the same number of live parity blocks and live index blocks; moreover, the j-th *live* index block corresponds to the j-th *live* parity block, for all j. The encoding procedure will make sure that this correspondence is preserved throughout the second stage: whenever a parity block becomes full, it will make the corresponding index block full as well.

We are now ready to present the encoding and decoding maps which are, again, specified in C-like pseudo-code notation.

Decoding map $\mathcal{D}_1$: The input to this map is a cell-state vector $\boldsymbol{x} = (\boldsymbol{x}_1 | \boldsymbol{x}_2 | \cdots | \boldsymbol{x}_{2m} \| \boldsymbol{u}_1 | \boldsymbol{u}_2 | \cdots | \boldsymbol{u}_{2k-2})$, partitioned into $2m$ parity blocks, of $k/2$ cells each, and $2(k-1)$ index blocks. The output is the information vector $(v_0, v_1, \ldots, v_{k-1})$.

```
(v_0, v_1, ..., v_{k-1}) = (0, 0, ..., 0);
for (ℓ = j = 1; j ⩽ 2m; j = j + 1)
{
  if (full(x_j)) continue; // skip full blocks
  while (full(u_ℓ)) ℓ = ℓ + 1; // skip full blocks
  i = u_ℓ; ℓ = ℓ + 1;
  if (i ≠ 0) v_{i-1} = parity(x_j);
}
```

Given an index i of the bit that has changed, the encoding map $\mathcal{E}_1$ first tries to find an active parity block $\boldsymbol{x}$ that represents the i-th information bit. If such a block is found, it is incremented and checked to see if it is full (in which case the corresponding index block is set to $q^\mu - 1$). If not, another live parity block is allocated to represent the i-th information bit. If no more live parity blocks are available,

the erasure symbol E is returned.

Encoding map $\mathcal{E}_1$: The input to this map is a cell-state vector $\boldsymbol{x} = (\boldsymbol{x}_1 | \boldsymbol{x}_2 | \cdots | \boldsymbol{x}_{2m} \| \boldsymbol{u}_1 | \boldsymbol{u}_2 | \cdots | \boldsymbol{u}_{2k-2})$, partitioned into $2m$ parity blocks and $2(k-1)$ index blocks, and an index i of the information bit that changed. Its output is either a cell-state vector $\boldsymbol{y} = (\boldsymbol{y}_1 | \boldsymbol{y}_2 | \cdots | \boldsymbol{y}_{2m} \| \boldsymbol{u}'_1 | \boldsymbol{u}'_2 | \cdots | \boldsymbol{u}'_{2k-2})$ or the symbol E.

```
(y_1 | y_2 | ··· | y_{2m}) = (x_1 | x_2 | ··· | x_{2m});
(u'_1 | u'_2 | ··· | u'_{2k-2}) = (u_1 | u_2 | ··· | u_{2k-2});
for (ℓ = j = 1; j ⩽ 2m; j = j + 1)
{
  if (full(x_j)) continue;
  while (full(u_ℓ)) ℓ = ℓ + 1;
  if (u_ℓ == i + 1)
  {
    increment(y_j);
    if(full(y_j)) u'_ℓ = q^μ − 1;
    break;
  }
  else ℓ = ℓ + 1;
}
if (j == 2m + 1) // active block not found
for (ℓ = j = 1; j ⩽ 2m; j = j + 1)
{
  if (full(x_j)) continue;
  while (full(u_ℓ)) ℓ = ℓ + 1;
  if (u_ℓ == 0)
  {
    u'_ℓ = i + 1;
    if (parity(x_j) ≠ v_i) increment(y_j);
    if (full(y_j)) u'_ℓ = q^μ − 1;
    break;
  }
  else ℓ = ℓ + 1;
}
if (j == 2m + 1) // no more available live blocks
return E;
```

Note that when the second encoding stage terminates, there are at most $k-1$ parity blocks that are not full, comprising at most $k(k-1)/2$ cells (at most $k(k-1)(q-1)/2$ cell-levels).

Once the maps $\mathcal{D}_1$ and $\mathcal{E}_1$ are understood, it becomes clear that the same approach can be applied iteratively. The resulting flash code $\mathbb{C}^*$ will proceed, sequentially, through s different encoding stages $\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_{s-1}$, where $s = \lceil \log_2 k \rceil$. In describing this code, we shall assume for the sake of simplicity that k is a power of two, that is $k = 2^s$. If not, the same code can be used to store $2^s > k$ information bits, of which the last $2^s - k$ are set to zero. Note that this will not change the order of the resulting write deficiency.

To accommodate the encoding maps $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_{s-1}$, we set aside for *each map* a batch of $2(k-1)$ index blocks, with each index block consisting of $\mu = \lceil \log_q(k+2) \rceil$ cells. The transition procedure $\mathcal{T}_r$ which bridges between the encoding maps $\mathcal{E}_{r-1}$ and $\mathcal{E}_r$ (for some $r \in \{2, 3, \ldots, s-1\}$) is identical to the transition procedure $\mathcal{T}_1$, except for the following differences:

D1. The r-th batch of index blocks is used; and

D2. The parity blocks consist of $k/2^r$ cells each.

In addition to **D1** and **D2**, the decoding/encoding maps $\mathcal{D}_r$ and $\mathcal{E}_r$ differ from $\mathcal{D}_1$ and $\mathcal{E}_1$ in that "$2m$" should be replaced by "$2^r m$" throughout, where m stands for $\lfloor n/k \rfloor$ as before. There are no other differences.

Theorem 3. *For $s = \lceil \log_2 k \rceil$, the write deficiency of the flash code $\mathbb{C}^*$ defined by the sequence of decoding/encoding maps $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{s-1}$ and $\mathcal{E}_o, \mathcal{E}_1, \ldots, \mathcal{E}_{s-1}$ is $O\big(qk \log^2 k / \log q\big)$.*

Proof. We consider the worst-case scenario for the number of cell levels that are either unused or "wasted" in the overall encoding procedure. As before, there are at most $k-1$ cells that are unused due to the partition into $\lfloor n/k \rfloor$ blocks, of exactly k cells each, at the very first encoding stage. These cells contribute at most $(q-1)(k-1)$ unused cell levels. The index blocks for the $s-1$ encoding maps $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_{s-1}$ contain $2(k-1)(s-1)\mu$ cells altogether, thereby wasting at most

$$2(q-1)(k-1)(s-1)\lceil \log_q(k+2) \rceil \;=\; O\left(\frac{qk \log^2 k}{\log q}\right) \quad (5)$$

cell levels. In each of the $s-1$ transition procedures, the situation $\texttt{parity}(x_i) \neq v_{i-1}$ can occur at most k times, and each time it occurs a single cell level is wasted. Finally, as in Theorem 2, when the encoding process $\mathcal{E}_o, \mathcal{E}_1, \ldots, \mathcal{E}_{s-1}$ terminates there are at most $k-1$ parity blocks that are not full and, in the worst case, each of them uses just one cell level. However, now these parity blocks contain only $\lceil k/2^{s-1} \rceil = 2$ cells each, and thus contribute at most $(k-1)(2q-3)$ unused cell levels. Putting all of this together, we find that at most

$$(q-1)(k-1)\Big(2(s-1)\lceil \log_q(k+2) \rceil \;+\; 3\Big) \;+\; k(s-1) \quad (6)$$

cell levels are wasted or left unused. Clearly, this expression is dominated by (5), and thus bounded by $O\big(qk \log^2 k / \log q\big)$. ∎

For large q, the upper bound of $O\big(qk \log^2 k / \log q\big)$ on the deficiency of our scheme can be improved by using a more efficient "packaging" of index blocks in the flash memory. As before, we allocate a batch of $2(k-1)$ index blocks to each encoding stage except $\mathcal{E}_0$. But now, every index block will occupy $\mu' = \lceil \log_2(k+2) \rceil$ cells rather than $\mu = \lceil \log_q(k+2) \rceil$ cells, and the indices will be written in binary rather than in the base-q number system. This allows index blocks that correspond to successive encoding stages to be "stacked on top of each other" in the same memory cells. Specifically, the encoding stage $\mathcal{E}_1$ will use only cell levels 0 and 1 to record the indices in its index blocks. Once this stage is over, the index information recorded during $\mathcal{T}_1$ and $\mathcal{E}_1$ is no longer relevant, and the level of *all* the $2(k-1)\mu'$ cells in the $2(k-1)$ index blocks can be raised to 1. Thereafter, provided $q \geqslant 3$, the transition procedure $\mathcal{T}_2$ and the encoding map $\mathcal{E}_2$ can use cell levels 1 and 2 to record the relevant index information in the *same memory cells*. Proceeding in this manner, we can accommodate up to $q-1$ batches of index blocks in $2(k-1)\mu'$ memory cells. We shall refer to this indexing scheme as *stacked binary indexing* and denote the resulting flash code by $\mathbb{C}'$.

Theorem 4. *The write deficiency of the flash code $\mathbb{C}'$ defined by the sequence of decoding/encoding maps $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{s-1}$ and $\mathcal{E}_o, \mathcal{E}_1, \ldots, \mathcal{E}_{s-1}$ that use stacked binary indexing is at most $O(qk \log k)$ if $q \geqslant \log_2 k$, and at most $O(k \log^2 k)$ otherwise.*

Proof. With stacked binary indexing, the number of cell levels wasted in all the $2(k-1)(s-1)$ index blocks is at most

$$2(q-1)(k-1)\left\lceil \frac{s-1}{q-1} \right\rceil \lceil \log_2(k+2) \rceil \quad (7)$$

Although for most values of k and q this is strictly less than (5), all the other terms in (6) are still dominated by (7). ∎

Remark. If we need to store k *symbols*, rather than bits, over an alphabet of size $\ell > 2$, the same flash code can still be used, with an appropriate interface. With the linear WOM-code of [17], the ℓ-ary symbols can be represented using $\ell - 1$ bits in such a way that any symbol change corresponds to a single bit transition. The flash code $\mathbb{C}'$ can be now applied as is, and the resulting write deficiency is $O\big(\max\{q, \log_2 k\ell\}\, k\ell \log k\ell\big)$.

VI. Flash Codes of Constant Rate

All of our results so far pertain to the case where $n \geqslant k^2$. In this section, we briefly examine the situation where both k and n are large, while $k/n = R$ for some constant $R < 1$. Observe that write deficiency $\delta(\mathbb{C}) = n(q-1) - t$ is *not* an appropriate figure of merit in this situation: a trivial code that guarantees $t = 0$ writes achieves write deficiency $n(q-1) = k(q-1)/R$, which is within a constant factor $2/R$ from the lower bound (1). Thus we will state our results in terms of the guaranteed number of writes t rather than the write deficiency $\delta(\mathbb{C})$.

If $q = 2$, we can easily guarantee $\Omega(n/\log k)$ writes as follows: partition the n cells into blocks of size $\lceil \log_2 k \rceil$ and each time an information bit changes, record its index in the next available block. For $q > 2$, the same method guarantees about $\lfloor n/\log_q k \rfloor = \Omega(n \log q / \log k)$ writes, but we can do better.

Let us partition the n cells into two groups: the *index group* consisting of $n-k$ cells and the *parity group* consisting of k cells. The index group is then subdivided into $m = \lfloor (n-k)/s \rfloor$ blocks, each consisting of $s = \lceil \log_2 k \rceil$ cells. The writing proceeds in $q-1$ phases. During the first phase, every time an information bit changes, its index is recorded in binary (using

cell levels 0 and 1) in the next available index block. After m writes, the first phase is over. We then copy the k information bits into the k cells of the parity group, and raise the level of all cells in the index group to 1. The second phase can now proceed using cell levels 1 and 2, and recording changes in information bits relative to the values stored in the parity group. At the end of the second phase, the current values of the k bits are recorded in the parity cells using levels 1 and 2, and so on. This simple coding scheme achieves

$$m(q-1) \;=\; \frac{n(q-1)(1-R)}{\log_2 k} \;=\; \Omega\left(\frac{nq}{\log k}\right) \qquad (8)$$

writes (where the middle expression ignores ceilings/floors by assuming that k is a power of two and that $n-k$ is divisible by $\log_2 k$). If q is odd and $R \geqslant 0.415$, we can do a little better by using the ternary number system (cell levels $0,1,2$) in both the index group and the parity group. In this case, the size of the parity group is $\lceil k/\log_2 3 \rceil$ cells and $1-R$ in (8) can be replaced by $(\log_2 3 - R)/2$. Finally, for all $R \geqslant 0.755$ and $q-1$ divisible by three, the quaternary alphabet is optimal, leading to a factor of $(2-R)/3$ rather than $1-R$ in (8).

VII. Buffer Codes

Buffer codes were first presented by Bohossian et al. in [1]. In this family of codes, a buffer of r symbols has to be stored in n flash memory q-ary cells. After each write, the last r symbols that were written have to be recovered by the cell-state vector. The goal is to maximize t, the number of write symbols that the code guarantees without incurring a block erasure. In [1], [10], an upper bound and a construction are presented for the case where the buffer is stored in a single cell. It is also shown how to store a buffer where, n, the number of cells satisfies $n \geqslant 2r$.

A. Buffer Codes Definition

We refer to the set of vectors in $\{0,\ldots,\ell-1\}^r$ as ***buffer vectors***. Similarly to a flash code, a buffer code $\mathbb{C}$ is also specified by an encoding map $\mathcal{E}$ and a decoding map $\mathcal{D}$. The ***decoding map*** $\mathcal{D}\colon \mathcal{A}_q^n \to \{0,\ldots,\ell-1\}^r$ assigns for each cell-state vector $\boldsymbol{x} \in \mathcal{A}_q^n$ its buffer vector $\mathcal{D}(\boldsymbol{x})$. The ***encoding map*** $\mathcal{E}\colon \{0,\ldots,\ell-1\} \times \mathcal{A}_q^n \to \mathcal{A}_q^n \cup \{\mathsf{E}\}$ specifies for every symbol $a \in \{0,\ldots,\ell-1\}$ and cell-state vector $\boldsymbol{x} \in \mathcal{A}_q^n$, another cell-state vector $\boldsymbol{y} = \mathcal{E}(a,\boldsymbol{x})$ such that $y_j \geqslant x_j$ for all $1 \leqslant j \leqslant n$, $(\mathcal{D}(\boldsymbol{y}))_1 = a$ and for $2 \leqslant i \leqslant r$, $(\mathcal{D}(\boldsymbol{y}))_i = (\mathcal{D}(\boldsymbol{x}))_{i-1}$. In case such a $\boldsymbol{y} \in \mathcal{A}_q^n$ does not exist, then $\mathcal{E}(i,\boldsymbol{x}) = \mathsf{E}$.

Definition. *An $(n,r,\ell,t)_q$ buffer code $\mathbb{C}(\mathcal{D},\mathcal{E})$ **guarantees** t **writes** if for all sequences of up to t symbol writes, the encoding map $\mathcal{E}$ does not produce the block erasure symbol* E.

B. Single-Cell Buffer Codes

In this section, we discuss the case where there is a single cell ($n=1$) to store the buffer. A construction for this scenario where a binary buffer ($\ell = 2$) is stored was given in [1], [10]. This construction guarantees at least $t = \left\lfloor \frac{q}{2^{r-1}} \right\rfloor + r - 2$ writes before a block erasure. An upper bound was given as well, which asserts that for every buffer code with one cell, the number of writes t has to satisfy

$$t \leqslant \left\lfloor \frac{q-1}{\ell^r - 1} \right\rfloor \cdot r + \lfloor ((q-1) \bmod (\ell^r - 1) + 1) \rfloor .$$

Let us show here another upper bound for such codes.

Theorem 5. *For any $(1,r,\ell,t)_q$ buffer code $\mathbb{C}$ such that $q \geqslant \ell^r$,*

$$t \leqslant \left\lfloor \frac{q-\ell^r}{\frac{1}{r}\sum_{d|r} \varphi(\frac{r}{d})\ell^d} \right\rfloor + r,$$

where φ is Euler's φ function.

Proof: Let $\mathbb{C}(\mathcal{D},\mathcal{E})$ be a $(1,r,\ell,t)_q$ buffer code. After $i \geqslant 1$ writes, for each $\boldsymbol{v} \in \{0,1,\ldots,\ell-1\}^r$, let

$$S_i(\boldsymbol{v}) = \{x \mid \text{there is a sequence of } j \leqslant i \text{ symbol writes ending in level } x \text{ and } \mathcal{D}(x) = \boldsymbol{v}\},$$

$m_i(\boldsymbol{v}) = \max_{x \in S_i(\boldsymbol{v})}\{x\}$ is the maximum cell level that is possible to reach after i symbol writes such that $\mathcal{D}(m_i(\boldsymbol{v})) = \boldsymbol{v}$, and

$$M_i = \sum_{\boldsymbol{v} \in \{0,\ldots,\ell-1\}^r} |S_i(\boldsymbol{v})|.$$

Clearly, for all $i \leqslant t$, $M_i \leqslant q-1$. After r writes, it is possible to reach any of the ℓ^r different buffer vectors and thus $M_r \geqslant \ell^r - 1$.

Let $\mathcal{G}_{\ell,r}$ be the r-th order ℓ-ary de Bruijn graph [3]. Its vertex set is $\mathcal{V}_{\ell,r} = \{0,1,\ldots,\ell-1\}^r$ and its edge set is $\mathcal{E}_{\ell,r}$. Let $\boldsymbol{v}_1, \boldsymbol{v}_2 \in \{0,1,\ldots,\ell-1\}^r$ be two different buffer states. Note that if $(\boldsymbol{v}_1,\boldsymbol{v}_2) \in \mathcal{E}_{\ell,r}$ and $m_i(\boldsymbol{v}_1) > m_i(\boldsymbol{v}_2)$ then $m_{i+1}(\boldsymbol{v}_2) > m_i(\boldsymbol{v}_2)$ and therefore, the value of M_{i+1} increases by at least one level for every such an edge. In the de Bruijn graph, every cycle has at least one edge $(\boldsymbol{v}_1,\boldsymbol{v}_2) \in \mathcal{E}_{\ell,r}$ such that $m_i(\boldsymbol{v}_1) > m_i(\boldsymbol{v}_2)$. Therefore, the number of new unused levels is at least the number of disjoint vertex cycles in $\mathcal{G}_{\ell,r}$. This number is known to be $\frac{1}{r}\sum_{d|r}\varphi(\frac{r}{d})\ell^d$ [7], [16], and therefore

$$t \leqslant \left\lfloor \frac{q-\ell^r}{\frac{1}{r}\sum_{d|r} \varphi(\frac{r}{d})\ell^d} \right\rfloor + r.$$

■

Lemma 6. *The bound in Theorem 5 improves the bound in [1] for $q \geqslant \ell^r$. That is,*

$$\left\lfloor \frac{q-\ell^r}{\frac{1}{r}\sum_{d|r} \varphi(\frac{r}{d})\ell^d} \right\rfloor + r \leqslant \left\lfloor \frac{q-1}{\ell^r - 1} \right\rfloor \cdot r + \left\lfloor \log_\ell \left(((q-1) \bmod (\ell^r - 1)) + 1 \right) \right\rfloor .$$

Proof: Note that

$$\frac{1}{r}\sum_{d|r} \varphi\left(\frac{r}{d}\right)\ell^d \geqslant \frac{\ell^r + \ell\varphi(r)}{r},$$

and therefore

$$\left\lfloor \frac{q-\ell^r}{\frac{1}{r}\sum_{d|r}\varphi(\frac{r}{d})\ell^d} \right\rfloor + r \leqslant \left\lfloor \frac{q-\ell^r}{\frac{\ell^r+\ell\varphi(r)}{r}} \right\rfloor + r$$
$$= \left\lfloor \frac{q-\ell^r}{\ell^r+\ell\varphi(r)} \cdot r \right\rfloor + r = \left\lfloor \frac{q+\ell\varphi(r)}{\ell^r+\ell\varphi(r)} \cdot r \right\rfloor.$$

If we denote $q-1 = x(\ell^r-1)+y$, where $0 \leqslant y \leqslant \ell^r-1$, then

$$\left\lfloor \frac{q-\ell^r}{\frac{1}{r}\sum_{d|r}\varphi(\frac{r}{d})\ell^d} \right\rfloor + r \leqslant \left\lfloor \frac{q+\ell\varphi(r)}{\ell^r+\ell\varphi(r)} \cdot r \right\rfloor$$
$$= \left\lfloor \frac{x(\ell^r-1)+y+1+\ell\varphi(r)}{\ell^r+\ell\varphi(r)} \cdot r \right\rfloor$$
$$= \left\lfloor \frac{x(\ell^r+\ell\varphi(r))-x+y+1-(x-1)\ell\varphi(r)}{\ell^r+\ell\varphi(r)} \cdot r \right\rfloor$$
$$= xr + \left\lfloor \frac{-x+y+1-(x-1)\ell\varphi(r)}{\ell^r+\ell\varphi(r)} \cdot r \right\rfloor$$
$$\leqslant xr + \left\lfloor \frac{(y+1)r}{\ell^r} \right\rfloor.$$

Let us show that $\frac{(y+1)r}{\ell^r} \leqslant \log_\ell(y+1)$. That is, we show that $(y+1) \geqslant \ell^{\frac{(y+1)r}{\ell^r}}$ or

$$\left((y+1)^{\frac{1}{y+1}}\right)^{\ell^r} \geqslant \ell^r.$$

The function $f(x) = x^{\frac{1}{x}}$ is monotonically decreasing for $x \geqslant 1$ and since $y \leqslant \ell^r-1$, we get

$$\left((y+1)^{\frac{1}{y+1}}\right)^{\ell^r} \geqslant \left((\ell^r)^{\frac{1}{\ell^r}}\right)^{\ell^r} = \ell^r.$$

Putting these together we get

$$\left\lfloor \frac{q-\ell^r}{\frac{1}{r}\sum_{d|r}\varphi(\frac{r}{d})\ell^d} \right\rfloor + r \leqslant xr + \left\lfloor \frac{(y+1)r}{\ell^r} \right\rfloor$$
$$\leqslant xr + \lfloor \log_\ell(y+1) \rfloor$$
$$= \left\lfloor \frac{q-1}{\ell^r-1} \right\rfloor \cdot r + \left\lfloor \log_\ell\left(((q-1) \bmod (\ell^r-1))+1\right) \right\rfloor.$$

■

C. Multiple-Cells Buffer Codes

In [1], [10], a buffer code construction is given for $\ell = 2$ and arbitrary n, q, r, where $n \geqslant 2r$. This construction guarantees $t = (q-1)(n-2r+1)+r-1$ writes. In this section, we show how to improve this construction such that the guaranteed number of writes is $t = (q-1)(n-r)$.

In the case where $q = 2$, the construction in [1], [10] guarantees $n-r$ writes. The encoding procedure is performed in such a way that after i writes, $1 \leqslant i \leqslant n-r$, the buffer is located between the $(i+1)$-st and $(i+r)$-th cells, where the first bit of the buffer memory is stored in the $(i+r)$-th cell and the last bit is stored in the $(i+1)$-st cell. If $q > 2$, then the construction uses a "layer by layer" approach. That is, first the layer of levels 0 and 1 is used, then the layer of levels 1 and 2 is used, and so on. In the transition from the layer of levels $i-1$ and i to the layer of levels i and $i+1$, all the cells are first reset to level i and the buffer is written in the new layer of levels i and $i+1$. Then, it is possible to continue writing in this layer. Basically, on each layer, it is possible to write $n-r$ times. However, when a new layer is used, then first the buffer from the previous layer is copied and then it is written in the new layer. Hence, it is possible to have only $(n-2r+1)$ more writes in the new layer and thus the total number of writes is

$$n-r+(q-2)(n-2r+1) = (q-1)(n-2r+1)+r-1.$$

The transition between these consecutive layers is not performed efficiently and our improvement here shows how it is possible to write $n-r$ times on each layer such that the total number of writes is $t = (q-1)(n-r)$. We first demonstrate how the construction works by the following example.

Example 2. In this example, we show how the last construction works for $n = 11, q = 3, \ell = 2$ and $r = 4$, so the number of writes is $2 \cdot (11-4) = 14$. The sequence of bits to be written is $1,1,0,0,1,0,0,1,1,1,0,1,1,0$ and the writes are performed as follows. The underlined cells represent the cells that store the buffer on each write.

Written Bit	Buffer State	Cell State Vector
	$(0,0,0,0)$	$(\underline{0,0,0,0},0,0,0,0,0,0,0,0)$
1	$(0,0,0,1)$	$(0,\underline{0,0,0,1},0,0,0,0,0,0,0)$
1	$(0,0,1,1)$	$(0,0,\underline{0,0,1,1},0,0,0,0,0,0)$
0	$(0,1,1,0)$	$(1,0,0,\underline{0,1,1,0},0,0,0,0,0)$
0	$(1,1,0,0)$	$(1,1,0,0,\underline{1,1,0,0},0,0,0,0)$
1	$(1,0,0,1)$	$(1,1,0,0,1,\underline{1,0,0,1},0,0)$
0	$(0,0,1,0)$	$(1,1,1,0,1,1,\underline{0,0,1,0},0)$
0	$(0,1,0,0)$	$(1,1,1,1,1,1,0,\underline{0,1,0,0})$
1	$(1,0,0,1)$	$(1,1,1,1,\underline{2},1,1,1,\underline{1,0,0})$
1	$(0,0,1,1)$	$(1,1,1,1,\underline{2,2},1,1,1,\underline{0,0})$
1	$(0,1,1,1)$	$(1,1,1,1,\underline{2,2,2},1,1,1,\underline{0})$
0	$(1,1,1,0)$	$(2,1,1,1,\underline{2,2,2,1},1,1,1)$
1	$(1,1,0,1)$	$(2,1,1,1,2,\underline{2,2,1,2},1,1)$
1	$(1,0,1,1)$	$(2,1,1,1,2,2,\underline{2,1,2,2},1)$
0	$(0,1,1,0)$	$(2,1,1,1,2,2,2,\underline{1,2,2,1})$

Now we are ready to present the construction by specifying its encoding and decoding maps specification.

Decoding map $\mathcal{D}_{\text{buf}}$: The input to this map is a cell-state vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$. The output is the corresponding information buffer vector $(v_1, v_2, \ldots, v_r)$.

```
m = max(x_1, x_2, ..., x_n);
n_m = find_repeat(m, x_1, x_2, ..., x_n);
if(n_m ⩾ r)
  for(i = 1; i ⩽ r; i = i + 1)
    v_i = x_{r+n_m-i+1} - m;
else {
  for(i = 1; i ⩽ n_m; i = i + 1)
    v_i = x_{r+n_m-i+1} - m;
  for(i = n_m + 1; i ⩽ r; i = i + 1)
    v_i = x_{n+n_m-i+1} - (m - 1);}
```

The function **max(**$x_1, x_2, \ldots, x_n$**)** simply returns the maximum value of the cells $x_1, x_2, \ldots, x_n$. The function **find_repeat(**$m, x_1, x_2, \ldots, x_n$**)** returns the number of times the value m repeats in the cells $x_1, x_2, \ldots, x_n$. If the value of n_m is at least r then the buffer is stored between the (n_m+1)-st and (n_m+r)-th cells, and the buffer values are calculated by subtracting m from the value of each cell. If the value of n_m is less than r then the buffer is stored cyclically in two cell groups: the last $r-n_m$ cells and the n_m cells in locations $r+1, \ldots, r+n_m$. In the first group, the buffer values are given by subtracting $m-1$ from the cells' value and in the second group by subtracting m from the cells' value.

Encoding map $\mathcal{E}_{\text{buf}}$: The input to this map is a cell-state vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$, and a new bit b. Its output is either a cell-state vector $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)$ or the erasure symbol E.

```
(y_1, y_2, ..., y_n) = (x_1, x_2, ..., x_n);
m = max(x_1, x_2, ..., x_n);
n_m = find_repeat(m, x_1, x_2, ..., x_n);
if(m == 0) {  // if this is the first write
  if(b == 1) y_{r+1} = 1;
  else  y_1 = 1; }
if(n_m == n - r) {  // first write in this layer
  for(i = 1; i ⩽ n - r + 1; i = i + 1)
    y_i = m;
  if(b == 1) y_{r+1} = m + 1;
  else  y_1 = m + 1; }
if(n_m < n - r) {  // not the first write in this layer
  y_{r+n_m+1} = y_{r+n_m+1} + b;
  if(b == 0)
    for(i = 1; i ⩽ n_m + r; i = i + 1)
      if(y_i == m - 1) {
        y_{r+n_m+1} = y_{r+n_m+1} + 1; break; } }
if(n_m ⩽ r - 1)  // one of first r - 1 writes in this layer
    y_{n-r+1+n_m} = m - 1;
```

On the first write, according to the bit value b, the first or the $(r+1)$-st cell changes its value to one. On the first write on each layer, the first $n-r+1$ cells are increased to level m, and then the first or the $(r+1)$-st cell is increased by one level, according to the bit value b. For all other writes, if the value if b is one then we simply increase the $(r+n_m+1)$-st cell by one level, and otherwise we increase the first cell of level $m-1$ by one level. Finally, if it is one of the first $r-1$ writes in each level, then we need to update the last cell that stores the buffer to level $m-1$ since it no longer stores the buffer and thus its level has to be updated.

Next, we prove the correctness of the construction.

Lemma 7. *After $s = x(n-r)+y$, where $1 \leqslant y \leqslant n-r$, the maximum cell level is $x+1$ and there are y cells in level $x+1$.*

Proof: According to the encoding map $\mathcal{E}_{\text{buf}}$, the maximum cell level increases every $n-r$ writes, on the $(i(n-r)+1)$-st write, for $0 \leqslant i \leqslant q-2$. Therefore, after s writes, the maximum cell value is $x = \lceil \frac{s}{n-r} \rceil$. If $y = 1$ then the maximum cell value is $x+1$ and we can see that exactly one cell changes its value to $x+1$. For all other writes, the maximum cell value does not change and exactly one cell changes its value to the maximum cell value which is $x+1$. ■

Theorem 8. *The buffer code $\mathbb{C}(\mathcal{D}_{buf}, \mathcal{E}_{buf})$ stores the buffer successfully and guarantees $t = (q-1)(n-r)$ writes.*

Proof: According to Lemma 7, after $t = (q-1)(n-r)$ writes the maximum cell level does not reach level q and hence there is no need to erase the block of cells. We prove the correctness of the encoding and decoding maps to store the correct value of the buffer by induction on the number of writes s. This is done by proving that for all $1 \leqslant s \leqslant t$, such that $s = x(n-r)+y$, where $1 \leqslant y \leqslant n-r$, the buffer $(v_1, \ldots, v_r)$ is calculated successfully according to the decoding rules of the decoding map:

1) If $y \geqslant r$ then for $1 \leqslant i \leqslant r$, $v_i = x_{r+y-i+1} - m$.
2) If $y < r$ then for $1 \leqslant i \leqslant y$, $v_i = x_{r+y-i+1} - m$ and for $y+1 \leqslant i \leqslant r$, $v_i = x_{r+y-i+1} - (m-1)$.

It is straightforward to verify that after the first write the memory successfully stores the buffer. Assume the assertion is correct after the s-th write, where $1 \leqslant s = x(n-r)+y \leqslant t-1$, $1 \leqslant y \leqslant n-r$. Assume that the new bit to be written to the buffer on the $(s+1)$-st write is b and let us consider the following cases:

1) If $y = n-r$, then on the $(s+1)$-st write in the encoding map the value of n_m is $n-r$. Thus the first $n-r+1$ cells change their value to $m = x$, the values of the last $r-1$ cells do not change, and if $b = 1$ then $y_{r+1} = m+1$, and otherwise $y_1 = m+1$. Therefore, the new value of the buffer is also given according to the decoding rules.
2) If $y < n-r$, then $n_m = y < n-r$, and the value of the $(r+n_m+1)$-st cell increases by b so the buffer is shifted one place to the right and it stores its updated value. If $b = 0$, then we increase the first n_m+1 cells by one level. Note that $n_m = y$ and there are exactly y cells with the maximum value so we can always find a cell of value less than m and increase the value to m. Then, the buffer is again stored according to the above decoding rules.

■

VIII. Conclusion

Rewriting codes for flash memories are important as they can increase the lifetime of the memory. Examples of such codes are flash codes [9] and buffer codes [1]. A significant contribution in this paper is an efficient construction of flash codes that support the storage of any number of bits. We show that the write deficiency order of the code is $O(k \log k \cdot \max\{\log_2 k, q\})$, which is an improvement upon the write deficiency order of the equivalent constructions in [10], [11], [19]. The upper bound in [9] on the guaranteed number of writes implies that the order of the lower bound on the deficiency is $O(kq)$. Therefore, there is a gap, which we believe can be reduced, between the write deficiency orders

of our construction and the lower bound. For buffer codes, we showed how to improve an upper bound on the number of writes in the case where one cell is used to store the buffer. If there are multiple cells, we showed a construction that improves upon the one presented in [1], [10].

REFERENCES

[1] V. Bohossian, A. Jiang, and J. Bruck, "Buffer coding for asymmetric multilevel memory," in *Proc. IEEE Int. Symp. Inf. Theory*, Nice, France, June 2007, pp. 1186–1190

[2] G.D. Cohen, P. Godlewski, and F. Merkx, "Linear binary code for write-once memories," *IEEE Trans. Inf. Theory*, vol. 32, no. 5, pp. 697–700, Sep. 1986.

[3] N.G. de Bruijn, "A combinatorial problem," in *Proc. K. Ned. Akad. Wet. Ser. A*, vol. 49, 1946, pp. 758–764.

[4] A. Fiat and A. Shamir, "Generalized write-once memories," *IEEE Trans. Inf. Theory*, vol. 30, pp. 470–480, Sep. 1984.

[5] H. Finucane, Z. Liu, and M. Mitzenmacher, "Designing floating codes for expected performance," in *Proc. 46th Ann. Allerton Conf. Commun., Contr. Comput.*, Monticello, IL, Sep. 2008, pp. 1389–1396.

[6] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," *ACM Computing Surveys*, vol. 37, pp. 138–163, June 2005.

[7] S.W. Golomb, *Shift Register Sequences*, revised edition, Aegean Park Press, Laguna Hills, CA, 1982.

[8] A. Jiang, "On the generalization of error-correcting WOM codes," in *Proc. IEEE Int. Symp. Inf. Theory*, Nice, France, June 2007, pp. 1391–1395.

[9] A. Jiang, V. Bohossian, and J. Bruck, "Floating codes for joint information storage in write asymmetric memories," in *Proc. IEEE Int. Symp. Inf. Theory*, Nice, France, June 2007, pp. 1166–1170.

[10] A. Jiang, V. Bohossian, and J. Bruck, "Rewriting codes for joint information storage in flash memories," *IEEE Trans. Inf. Theory*, vol. 56, no. 10, pp. 5300–5313, Oct. 2010.

[11] A. Jiang and J. Bruck, "Joint coding for flash memory storage," in *Proc. IEEE Int. Symp. Inf. Theory*, Toronto, Canada, July 2008, pp. 1741–1745.

[12] A. Jiang, M. Landberg, M. Schwartz, and J. Bruck, "Universal rewriting in constrained memories," in *Proc. IEEE Int. Symp. Inf. Theory*, Seoul, Korea, Jul. 2009, pp. 1219–1223.

[13] A. Jiang, R. Mateescu, M. Schwartz, and J. Bruck, "Rank modulation for flash memories," *IEEE Trans. Inf. Theory*, vol. 55, no. 6, pp. 2659–2673, Oct. 2010.

[14] A. Jiang, M. Schwartz, and J. Bruck, "Correcting charge-constrained errors in the rank modulation scheme," *IEEE Trans. Inf. Theory*, vol. 56, no. 5, pp. 2112–2120, May 2010.

[15] H. Mahdavifar, P.H. Siegel, A. Vardy, J.K. Wolf, and E. Yaakobi, "A nearly optimal construction of flash codes," in *Proc. IEEE Int. Symp. Inf. Theory*, Seoul, Korea, Jul. 2009, pp. 1239–1243.

[16] J. Mykkeltveit, "A proof of Golomb's conjecture for the de Bruijn graph", *Journal of Combinatorial Theory (B)*, vol. 13, pp. 40–45, 1972.

[17] R.L. Rivest and A. Shamir, "How to reuse a write-once memory," *Inf. Contr.*, vol. 55, no. 1–3, pp. 1–19, Dec. 1982.

[18] E. Yaakobi, P.H. Siegel, and J.K. Wolf, "Buffer codes for multi-level flash memory," presented at the *IEEE Int. Symp. Inf. Theory*, poster session, Toronto, Canada, July 2008.

[19] E. Yaakobi, A. Vardy, P.H. Siegel, and J.K. Wolf, "Multidimensional flash codes," in *Proc. 46th Ann. Allerton Conf. Commun., Contr. Comput.*, Monticello, IL, Sep. 2008, pp. 392–399.